4

ERL-0490-TR

AR-005-948

DEPARTMENT OF DEFENCE

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION
SALISBURY

ELECTRONICS RESEARCH LABORATORY

SOUTH AUSTRALIA

TECHNICAL REPORT

ERL-0490-TR

SOME COMMENTS ON TECHNIQUES FOR RESISTING COMPUTER VIRUSES

M. ANDERSON

COPY No.

AR-005-948

DEPARTMENT OF DEFENCE

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION

ELECTRONICS RESEARCH LABORATORY

TECHNICAL REPORT

ERL-0490-TR

SOME COMMENTS ON TECHNIQUES FOR RESISTING COMPUTER VIRUSES

M. Anderson

SUMMARY

Computer viruses have caused much concern. A basic flaw in the protection mechanism of most computer systems makes them very susceptible to viral attack. Various techniques for resisting viruses are discussed and their comparative advantages and disadvantages examined.

POSTAL ADDRESS: Director, Electronics Research Laboratory, PO Box 1600, Salisbury, South Australia 5108

TABLE OF CONTENTS

Page

1. INTRODUCTION 1

2. VIRUSES 2

3. A NAIVE PROTECTION TECHNIQUE 3

4. TOWARDS BROAD SPECTRUM PROTECTION 4

4.1 Protecting the operating system 4

4.2 Protecting the user 5
4.2.1 Encryption and checksums 5
4.2.2 Risk management 6
4.2.3 Limited integrity 7
4.2.4 Capabilities 9
4.2.5 Cohen's access program 10

5. CONCLUSION 11

REFERENCES 13

TABLE 1. CREDIBILITY VALUES 6

APPENDIX I SPECIFICATION FOR LIMITED INTEGRITY MECHANISM 15

LIST OF FIGURES

1. Flow control of a virus 16

2. Limited integrity 17

3. Invocation methods 18

1. INTRODUCTION

Computer viruses have caused concern in the information technology community. The basic protection mechanism of most machines is inadequate to deal with the virus problem, thus leaving them open to widespread corruption of data. The flaw in their protection mechanism is of such a fundamental nature even systems implementing multilevel information flow control (Cohen, 1988) can have files corrupted. The problem involves the fact that in many systems a program inherits all the rights and privileges of its invoker.

Due to the increasing interconnectivity of networks and also the use of imported software, tracking down the perpetrators of virus attacks can be extremely difficult. So far, one attitude taken by some installations to achieve "vaccination" from computer viruses is to disconnect their system from networks altogether. While this does provide protection at the network level, it still does not prevent the local user base from infecting the system. Also, the functionality of the system itself can be severely impaired by being cut off from what could be perceived as a necessary information transfer medium. Of course two separate systems with no interconnection could be supplied, one to handle network functions and the other to handle the everyday business of the nervous user base. However, two problems with this approach are the expense involved in the second system and irritable users having to log onto separate machines according to function. Isolating a machine from a network is not the answer.

This paper is concerned with describing and briefly contrasting various mechanisms that have been proposed to deal with the virus problem. As shall be seen, each mechanism has advantages and disadvantages when compared to others. There is no attempt to claim that the techniques described here will provide a watertight defence against viruses. Nevertheless, they do provide a useful resistance to the virus problem for varying degrees of effort. Note that we are interested in systems which are shared, have multistate protection hardware[1], and the authorised user base may be the source of viruses. Most personal computers and transaction systems do not satisfy all of these criteria.

We describe the salient characteristics of a virus. Then, we discuss a commonsense practice which can prevent the kernel[2] and operating system from being infected by any virus imported via a network or the user base. Finally, mechanisms are described which can help prevent a user's programs from being infected; some with surprisingly little loss of functionality in the system as a whole.

---

[1] For example, a system may have hardware which is able to run in "supervisor" or "user" mode where certain operations and privileges are either enabled, or disabled.

[2] That part of the Operating System which contains trusted code. The kernel is protected from unauthorised modification in memory by the hardware.

2. VIRUSES

Essentially, a computer virus is a code fragment which is implanted in a program to alter that program's function. For example assume the existence of some active virus program. The virus first searches for an executable file that is writeable. It then appends a copy of itself to some part of the code section, adds in a jump instruction to the usual entry point of the program, and then modifies that entry point to the virus code. Entry point addresses for programs in many computer systems can be obtained from what is generally dubbed the load module header residing in a known place of an executable file. The next time the program is invoked it ends up in the virus code which can then decide to reproduce further or to do something nasty (trigger). Figure 1 describes the basic flow control of many viruses. Variants on the virus theme include ones which modify their code structure before implantation or shift their position in the executable file. These tactics are intended to avoid detection by a program designed to track down the virus. Other types of viruses include those which are implanted on disks and are activated by the bootstrap programs on PC's.

A virus, for maximum effect, attempts to propagate as much as possible before triggering. It is not much fun for the virus programmer to ensnare only one or two users. A virus is much more devastating if it can quietly propagate to as many programs as possible in as many systems as possible and trigger more or less simultaneously. This can maximise the amount of damage done as there is some lead time before system administrators can react and attempt to minimise damage by radical methods such as shutting down the system.

For example, consider a virus which triggers at a given date and time. Assume that the virus is capable of, and indeed does, migrate between systems in a network. If the trigger date is close to the date the virus is first introduced then when the trigger condition occurs, and damage to systems becomes evident, there may be many systems still in the network which are unaffected. All it takes are some phone calls from the system administrator to others warning of the infection for them to isolate their machines from the net. Conversely, the system administrator with the infected machine can cut off its network communications as a community service. If the virus had time to propagate to many systems before its trigger announced its presence, the damage done can be much more severe and widespread.

The above premise for restricting infection is only valid among some forms of network. Networks are becoming increasingly transparent, especially Local Area Networks (LANS), and hence a virus is quite capable of spreading too rapidly for disconnection to prevent infection in other systems. In addition, the damage caused by a virus may be subtle enough to escape detection for some time. However, it should be noted that machines connected to long haul nets have been disconnected in time during past instances of viral-like attacks (see below).

A distinction is drawn between viruses and *worms*. A worm is a program in its own right. It usually gains entrance to a system by exploiting a trojan horse or some design flaw. Once in the system worms reproduce by creating copies of themselves and causing the system to invoke them as processes. Worms are much more easily detectable than viruses as they do not attempt to appear as a fragment of another, probably legitimate, program (host). It was a worm program which invaded various UNIX[3] systems in the US in 1988 (Spafford, 1988). The worm exploited an unintended "feature" of the mailer to gain maximum privileges. Other aspects included attempts to break passwords according to the age old method of looking up the system dictionary. It also tried quantities of "sweetheart" names. Surprisingly, the worm didn't appear to try expletives; a very common form of password in some UNIX communities. Communication (other than via the net) between system administrators prevented a number of systems from being infected. Apparently there was enough time for some systems to be disconnected from the net in order to prevent infection. However, it is sobering to realise that a large number of very dispersed systems were rapidly infected.

The given reference goes into great detail concerning the "breakin" on the internet and gives a detailed analysis of the type of person that may be responsible for the worm's construction.

An article by Witten (1987) goes into more detail regarding the structure of viruses and their characteristics.

3. A NAIVE PROTECTION TECHNIQUE

Obviously, most users wish to avoid having their programs infected with a virus. There's no telling what will happen to their data if one does get into their files and waits for some event to trigger. One might think that it is feasible to develop specific *antibodies*, ie programs which detect the virus and eliminate it. These antibodies are tailored to recognise the code sequence of a virus in a program and either flag it, or remove it from the program. Clearly, the sooner a system administrator detects a virus the sooner an antibody can be tailored to detect more instances of the virus and remove it.

Unfortunately, the same problem present in the human immune system becomes apparent in the computer system. The virus programmer can make simple changes to the virus so that it is no longer recognised by the antibody. The end result is that the luckless programmers assigned to prevent infection of a system must detect the presence of essentially the same virus and build a modified antibody to destroy it. Another problem is that unless all data in a system is checked, including backups, users may leave themselves open for reinfection at a later date. Somewhat ironically, even if a file does contain the virus sequence it may be untouchable due to its status as a data file. The sequence could possibly be legitimate as data but unlikely. In this case it is a good idea to keep an eye on the owner of the file. Yet another problem is that scanning all the programs in a system for different viruses can be very expensive in terms of cpu and I/O transfers.

One might claim that antibodies capable of detecting a family of viruses can be constructed. The antibody tests for code sequences similar to that of the base virus code sequence. There is a major difficulty in following this tack as the possible combinations of code sequences for a virus could be very large. Although

3 UNIX is trademark of AT&T

a combination generator can be constructed there is no guarantee it will hit on all of them. Besides, the more combinations there are to test the more cpu time is required.

4. TOWARDS BROAD SPECTRUM PROTECTION

One vital ability required by a virus is propagation. Without it, the virus is limited to inflicting damage whenever its host (the program in which it resides) is invoked. As soon as it does cause damage, and the damage detected, the program can be removed from the arena of use. With no reproduction capability the system users can have some relief from the knowledge that it is containable. Once removed, the virus no longer presents a problem. Given this, we conclude that a virus can be severely hampered if it is prevented from infecting files which are, or may become, executable. Indeed, if an entire system is *clean* and somehow all viruses could be prevented from infecting executable files, present and future, then the system is immune. Unfortunately, this utopia is not achievable in the full sense (Cohen, 1988). Nevertheless, we will discuss some methods which can go a long way to satisfying our goal so that if infection does occur, it can be contained. In the following discussion we will borrow from the UNIX terminology. However, unless stated otherwise, the techniques should be applicable in a wide range of timesharing systems.

4.1 Protecting the operating system

For system programs and utilities, prevention of infection is achieved relatively easily by simply placing the programs on a set of disks with the write protect switch set. An additional constraint is that no executable file can be created by the root[4] unless it is at a specified terminal. Naturally the files specifying which terminals are on the list are on the protected disk. Given these circumstances if the superuser[5] invokes a program with a virus at a terminal other than one on the list, the virus is limited to corrupting data files. It cannot hide itself in a system utility such as a compiler and thus spread itself. While this is cold comfort for the hapless users who have their files corrupted, the system administrator can rest assured that the operating system and all executables are clean. It should be remembered that system administrators are expected to exercise much more care than the average user when executing programs. Indeed, there should be a policy of not executing any program with superuser privilege without some form of verification or indication of trust in the software.

The above technique for keeping the operating system clean is suitable as long as the utilities are not changed too frequently. If a utility must be changed, then booting the system as single user and using only the current utilities to create the new one is a safe practice. Of course the write protect on the utility disks must be released while changing the requisite utility. Unfortunately, frequent shutting down of the system to change the utilities could very well irritate the user base. Hence it is desirable that changing a utility is performed at prudent intervals.

4 User or process with maximum privileges.

5 User with root privileges.

4.2 Protecting the user

We have described a simple method for protecting, or vaccinating, the system and its utilities from attacks by viruses. We now turn our attention to methods for protecting the user.

Again, the basic solution lies with the ability, or rather inability, to write and create files which may become at some stage executable. Source and intermediate code are included in this definition.

If a user executes a program, then typically it inherits that user's rights. If the executed program contains any virus code then the invoker is in trouble. The virus code has the same access privileges as the invoker and thus can gain access to any file accessible to the invoker. It is this type of protection mechanism that has allowed the virus problem to be as large as it is. As will be seen later, systems which do not have this flaw, and thus have the ability to be very resistant to viruses, have been developed for some time. However, let us concentrate for the moment on how to alleviate the problem given the situation where a program inherits a user's rights on invocation.

4.2.1 Encryption and checksums

It has been suggested that executable programs can be encrypted to protect them from unauthorised modification (Pozzo and Gray, 1987). When a program is compiled and linked, the binary executable is encrypted by some piece of trusted software with a secret key. Every time a program is run it is first decrypted and if the file has not be tampered with, it should proceed. Unauthorised modifications to the encrypted binary should produce rubbish for the computer to attempt to run.

Encrypting a binary executable every time it is created may be tolerable but decrypting a program every time it is run can be time consuming and may not be tolerable. The performance penalty, especially with public key encryption and large programs, tends to make it an unpalatable solution. It should be noted however that with the addition of relatively inexpensive dedicated coprocessors for encryption/decryption, some of the performance penalty may be removed.

Another related method is to place an encrypted checksum or digital signature in an executable. Any attempt to modify the given program without successfully recalculating a new signature and appending it to the file will result in detection at runtime. Although essentially all of a program must be scanned in calculating a checksum, this technique can be more efficient than if one had to encrypt/decrypt an entire file. The effort in calculating a reasonably strong encrypted checksum can be quite different to the amount of effort required in having to apply a strong encryption process to the entire contents of a file.

A disadvantage with either of the encryption mechanisms as described by Pozzo and Gray is that it is only applied to executable files. Source files and intermediate files representing object code before compiling or linking are still vulnerable to virus attack.

4.2.2 *Risk management*

Pozzo and Gray (1987) have also suggested the introduction of a risk management mechanism to help defeat the propagation of viruses and which can go some way to helping overcome the aforementioned disadvantage with intermediate files. Essentially, the mechanism requires programs to be assigned a "credibility" value by the system administrator with integers in the range 0-N. Software with a value of zero is considered to have the lowest credibility while software with the value of N is considered to have a high credibility.

The next part of the mechanism involves a "risk" value, in the range 0-N, which is associated with every process. A process inherits its risk value from its parent and ultimately, from a user who nominates a session risk value at login.

The risk mechanism ensures that no process with a risk level M can execute any software with a credibility value less than M unless the user explicitly overrides it with a command such as "run untrusted". Accordingly, virus propagation can be hampered if suspect programs are assigned low credibility values and users log in with higher risk values. Table 1 shows Pozzo and Gray's suggestions for credibility assignations by the system administrator.

TABLE 1. CREDIBILITY VALUES

Software type	Value
User files	0
User contributed SW	1
Bulletin Board SW	2
Commercial SW	3
SW from system staff	3
Verified SW	4
SW from OS vendor	5

Assuming that only system utilities with a high credibility level will be required for the manipulation of source and object code files, then in conjunction with the encryption mechanism a strong resistance to viruses is presented. A virus cannot infect source or object code unless a user more or less deliberately risks infection. Any infection of a binary executable is detected at runtime.

One disadvantage of the risk mechanism is that users cannot specify their own trust in items of software. Indeed, it would appear that users logged in with a high risk value could very well be unable to run their own programs unless they specify explicitly to run them untrusted. Groups of users may have a certain amount of trust in one another but have a lower amount of trust in another group. Users would no doubt wish to specify what programs they do or do not trust rather than rely solely on the system administrator. Of course they probably have to trust the utilities and system software, but there will certainly be a number of them wishing to be able to indicate levels of credibility amongst themselves; something the system administrator may not be qualified to do or in a position to handle efficiently.

4.2.3 Limited integrity

We describe a mechanism called *limited integrity* and then contrast it with the risk/encryption mechanisms. It is assumed that systems are able to specify any combination of read, write, and execute access to files for the owner of a file. Appendix I contains a more formal description of the limited integrity mechanism.

First, let there be a bit associated with every process called the *trust* bit. Also with every file let there be an associated *integrity* bit. We assume the bits can only be modified by trusted kernel software.

Let there be two forms of program invocation called exec_trust and exec respectively. If a process invokes a program, then unless it is of the type exec_trust, the system will clear the process's trust bit. Any process which spawns another must have its trust bit set for the spawned process to have a set trust bit. Without a set trust bit, the system will not allow a process to set the executable permission bit of a file and if a file with a set execute permission bit is opened with modify access, the execute bit is cleared. Naturally all other access rules also apply. So, if suspect programs do not have their trust bit set a virus will have a hard time propagating through executable images. Note that this does not really hamper a legitimate but suspect program's need to access other files. Of course an untrusted process can gain write access to its invoker's executables by changing the permissions of the requisite file to no execute and writeable, but this will lead to detection as the invoker will find out the file is mysteriously no longer executable. The untrusted process is unable to reset the executable bit. As will be seen below, the invoker has yet another method for detecting modification.

The above mechanism prevents undetected infection of executable files if the suspect programs are run without trust. However, it does not prevent a virus from manipulating a source or object code file and these are fairly easily recognised by the virus. The integrity bit can be used to overcome this deficiency. When a file is created[6], its integrity bit is set if the creating process has its trust bit set. If any untrusted process opens the file for writing, the integrity bit is cleared. Hence if a user ensures that all files forming a program development chain are created and manipulated by trusted processes only, eg system editors and compilers, any "unauthorised" modification can be

6 Renaming a file is considered to be a variant of creation.

detected. Figure 2 depicts the setting and clearing of integrity bits when various operations are performed with trusted and untrusted processes.[7]

Most application programs do not require a set trust bit. Probably the most frequently used applications which do require it are system utilities such as compilers and editors. A compiler can use the integrity bit of a file to see if it is the result of a trusted editing process before including it in a program.

If a virus is imported into the system it cannot propagate unless trust bits are set. As previously mentioned, many application programs do not require the setting of a trust bit so they should be able to fulfill their function, including the reading and writing of files, without it.

The limited integrity mechanism is much more efficient than the risk/encryption mechanisms. The trust bit is only set or cleared on program invocation and the integrity bit need only be modified when a file is opened for writing. The data structures holding these bits must be examined in any case and there is negligible additional overhead in checking one bit.

A disadvantage of the limited integrity mechanism is that it has to be implemented at the kernel level. However, it is claimed that the implementation is relatively simple. All that is required is to add a bit to the system data structure containing information on a file and to add a bit to the system information on a process. The points at which the bits must be examined or modified are well defined.

Another disadvantage of the limited integrity mechanism is that it cannot specify levels of trust between users. One either elects to trust a program and all other programs which may subsequently be invoked from that program or one does not. The mechanism does not specify multilevel integrity as in Biba's (1977) model.

It is interesting to note that Kuchta (1989) has released a preliminary description of the Canadian Trusted Computer Product Evaluation Criteria. It is a Canadian version of the "Orange Book" (Department of Defence, 1983). One of the aspects which distinguish it from the Orange Book is the taking into account of integrity. Kuchta describes support for mandatory and discretionary execution controls on programs depending on a perception of their trustworthiness or integrity to perform the required task. The point here is that given the Canadian criteria, the risk/encryption mechanism can be seen as a systemwide, mandatory integrity policy, whereas limited integrity can be seen as a restricted discretionary integrity policy. It is reasonable to expect that more integrity policies which help prevent viruses will appear as a side effect of efforts to formulate general integrity mechanisms. Further discussion on integrity, while of importance, is outside the scope of this paper.

---

7 The operations pertaining to setting and clearing execute permission bits are not shown.

4.2.4 Capabilities

Capabilities are like "tickets" which authorise their holder, typically processes, to gain access to objects, which are generally in the form of files. If a process does not hold a capability to a file, it cannot gain access to that file. Capabilities also describe what type of access is allowed to the file and there can be more than one capability for the file. For example, user A may have a capability to file X with read and write privilege, while B could also have a capability to file X with only read privilege. Also, most implementations provide protected mechanisms for the creation and destruction of capabilities. For example, privileges pending, A could make another capability to X which has only the read privilege and give it, say, to user C. At some stage A may destroy the capability given to C thus revoking C's access to X. Capabilities are generally implemented via some form of unforgeable, unique pointer which appears meaningless to a user. Hence directories associating ascii strings (read filenames) with capabilities are typically added. When a user specifies a filename the command interpreter can inspect a directory to obtain the requisite capability and pass this to programs requiring it.

We do not wish to go into detail regarding the implementation of capabilities as there are many facets and implications. Hence we merely proffer some references (Wulf, Levin, Harbison, 1979; Wilkes and Needham, 1979; Levy, 1984; Anderson and Wallace, 1988) for further reading.

Capabilities have been previously suggested as a strong, broad spectrum vaccine for computer viruses (Anderson, 1988b). Given a system based on capabilities, a user merely ensures that on invocation of a suspect program only capabilities to those files to which the program must gain access are given. Typically, one can envisage read access to some input files, and write access to some output files. If the program contains a virus it can only corrupt those files to which it has been given explicit write access. Hence the virus is severely hampered in its operations and if those output files are never part of the program development chain, propagation of the virus is stopped. In fact, any malicious program can be restrained in the amount of damage it can do.

Ensuring that a program at any one moment has access only to the objects required to complete its immediate task is called the "principle of minimum privilege".

It is no easy task to retrofit a capability based protection mechanism to existing computer systems. Certainly the kernel of the operating system has to be substantially modified. Since "native" capability based systems are not very popular[8] as yet, we shall not see an immediate cessation of the virus problem via their use. Hence the aforementioned mechanisms are still valid contenders to aid in virus restraint. However, it is worthwhile to note the ease with which the concept of capabilities and the principle of minimum privilege can place severe constraints on viruses.

---

8 A commercial system which has capability-like properties is the IBM System/38. Another capability based system which has been used for defence applications in the UK for quite some time is the Plessey 250.

4.2.5 Cohen's access program

Cohen (1988) devised a program which could restrict a program's access to particular files. He did this by exploiting the *setuid*[9] and by employing the client-server paradigm. Figure 3 helps depict what happens when a client has two files (file1, file2) from which a program is to take data, process it, and place the output in file3. Two implementations of the *access* program are shown. File3 must belong to the client at service termination. The figure also depicts what is involved when the other virus protection mechanisms are used. This example has been deliberately chosen because it is simple, common, and it shows an unfortunate level of complexity in *access*.

In figure 3(a) user A executes XYZ with the given parameters:

- XYZ - Name of program owned by client which opens up the input and output files and passes along file descriptors to the next program invoked, ie *access*.
- ABC - Name of service program owned by provider (B) of service.
- File1 - Name of input file owned by client.
- File2 - Name of input file owned by client.
- File3 - Name of output file to be owned by client after processing.

XYZ opens up the input and output files to obtain file descriptors, it then invokes *access* which looks in an *access control* list owned by B to see if the user request is justified. The access control list contains tuples comprising the uid of the user able to call on a service, the name of the service, and the programs invoked to carry out the service. For clarity, figure 3 depicts the name of the service and the name of the program actually invoked as the same.

If the request is justified, the service program is invoked with the process's uid set to that of the service owner. Because file descriptors can remain open across exec calls, all the programs in the chain can access the requisite files. The service program carries out the request and exits. The first difficulty raised here is that the file descriptors, similar in function to logical unit numbers in fortran programs, are only program local names for the files involved. Hence both the client and the server must agree on what file descriptors can be used and in what order. In a capability based system (figure 3(c)) this is not a problem as capabilities are unique, unambiguous pointers to the file to which access is granted[10].

Of course it is possible to pass the pathnames of the files to *access* and avoid the need for the XYZ program. However, all is not as straightforward as it may at first seem. Consider figure 3(b)). Here the pathnames have been issued to *access* and we find that unless the correct access permissions are set, eg write access to one of A's directories, File3 cannot be created. Moreover, if it was created it would belong to B rather than A. If the permissions allow the server access to one of A's directories, then it could well leave access to other users that A wishes to restrict severely.

9 It is assumed that the reader is familiar with most UNIX terminology in the UNIX system.

10 The * indicates that the command interpreter is to pass capabilities to the program.

This kind of situation can arise when the server is not in any of A's groups. To overcome this difficulty, *access* could be called a second time so that it invokes a program of A's which copies file3. Alternatively, A could set up an "unreadable" directory with a file in it with write permission granted to the server. Essentially, protection is achieved for the file by no one other than the client and the server knowing the file's name. The advantage of a program inheriting its invoker's rights is that it can process the invoker's files without the need for the invoker to manipulate access permissions. Thus files need not become visible to users other than the invoker and the program's owner. *Access* removes this advantage.

Two strong advantages of Cohen's access program are that it can be implemented easily at the user level and it has the ability to limit rights propagation at the process level. However, there are also some disadvantages. The mechanism relies on exploiting a feature (setuid) of a particular system. There is no intuitive assurance whatsoever it can be implemented easily on other systems as with the risk/encryption and limited integrity mechanisms. Secondly, programs which are invoked via *access* must follow some convention so that the local naming problem with file descriptors is avoided. Even if full pathnames are used programs of the client may have to be invoked, or some convention followed, to achieve the desired result. Thirdly, large amounts of existing software would have to be readjusted to follow *access's* conventions. The limited integrity and risk/encryption management schemes on the other hand, would have little impact on existing software. Note that with a capability based system, existing software would also have to be modified substantially.

5. CONCLUSION

By placing utilities on a set of write-protected disks they can be protected for little effort. Protecting the user on the other hand, is not immediately achieved. A capability based system can certainly make it extremely difficult for a virus to thrive. However, it is not a trivial task to implement such a mechanism on existing systems and capability based systems are not in plentiful supply.

The virus problem will never go away as long as people execute other people's programs without checking the program's integrity. However, it is possible to contain the amount of damage that a suspect program can do by varying amounts depending on the mechanism employed. The capability based approach offers the most rigorous protection, but requires significant effort in its implementation. Cohen's access program is easily implemented on a particular system and offers a wide range of access policies but would impact on existing software which does not follow some convention. The limited integrity scheme offers protection against viruses but cannot offer the same range of access policies. The mechanism is however easily implemented across a wide range of systems. The risk/encryption mechanisms also offer protection.

So, which is best?

It is suggested that the various mechanisms should be employed depending on the time frame for fitting virus protection. For the immediate future, the limited integrity and risk/encryption mechanisms are suitable as they offer protection for relatively little effort and disturb existing software the least. For UNIX systems, the access program should also be implemented and as time goes by more and more software can be accessed through it. In the long run however, capability based systems should be used as they can constrain processes in very small domains of protection. There are capability based systems which can constrain separate procedures of a program to accessing parts of files. This can place strong bonds on malicious code. Ultimately, systems which have automatic user rights inheritance must be phased out.

It is important to realise that the techniques described do not vaccinate a system from all types of viruses. If a virus is contained on a disk which is integrated into the operating system, or the kernel itself already contains a virus then trouble will ensue. Note that these types of viruses are generally the result of either bad system management practices, or inadequate hardware/kernel protection. There also may be other types of viruses, more subtle, which may be able to infect the memory image of a program while it is being run and gathering information. However, this is generally the result of not protecting the code segments properly from modification and poor code design.

REFERENCES

No.	Author	Title
1	Anderson, M. and Wallace, C.S.	"Some Comments on the Implementation of Capabilities". The Australian Computer Journal Vol 20, 3. pp 122-133, 1988
2	Anderson, M.	"A Note on Security and Unix". AUUGN, Vol 9, 2, 1988b
3	Biba, K.	"Integrity Considerations for Secure Computer Systems". USAF Electronic Systems Division, ESD-TR-76-372, 1977
4	Cohen, F.	"Computer Viruses: Implication and Methods of Defence". Computers and Security, 7 pp167-184, 1988
5	Department of Defence	"Department of Defense Trusted Computer System Evaluation Criteria". CSC-STD-001-83, 1983
6	Kuchta, M.	"The Canadian Trusted Computer System Evaluation Criteria". Proc. First Annual Computer Security Conference 1989
7	Levy, H.	"Capability Based Computer Systems". Digital Press. Burlington, MA, 1984
8	Pozzo, M. and Gray, T.	"An Approach to Containing Computer Viruses". Computers and Security, Vol 6, 4. pp321-331, 1987
9	Spafford, E.H.	"The Internet Worm Program: An Analysis". Purdue Technical Report CSD-TR-823, 1988
10	Wilkes, M.V. and Needham, R.M.	"The Cambridge CAP Computer and its Operating System". Elsevier North Holland
11	Witten, I.	"Computer (In)security: Infiltrating Open Systems". Abacus, Vol 4, 4. pp7-25, 1987

12	Wulf, W., Levin, R. and Harbison, S.P.	"HYDRA/C.mmp: An Experimental Computer System". McGraw-Hill, 1981

APPENDIX I

SPECIFICATION FOR LIMITED INTEGRITY MECHANISM

Below is a more formal description of the limited integrity mechanism than that given in the body of the paper.

CT refers to the current value of the trust bit of a process. CI refers to the current value of the integrity bit of a file. NT refers to the value of the trust bit after some operation has been performed, possibly involving CT. NI refers to the value of the integrity bit after some operation has been performed.

1. When a file is to be opened with modify access:

```
if file has execute permission then
    if not CT then
        remove execute permission
        NI = zero
else
    NI = CT and CI
```

2. If the execute permission bit of a file is to be set:

```
if not CT then
    abort operation
```

3. On process creation: let CT1 refer to the trust bit of the parent process, NT refer to the trust bit of the child.

```
NT = CT1
```

4. If executing a program with trust, ie exec_trust:

```
NT = CT
```

5. If executing a program without trust:

```
NT = zero
```

6. On file creation or renaming:

```
NI = CT
```

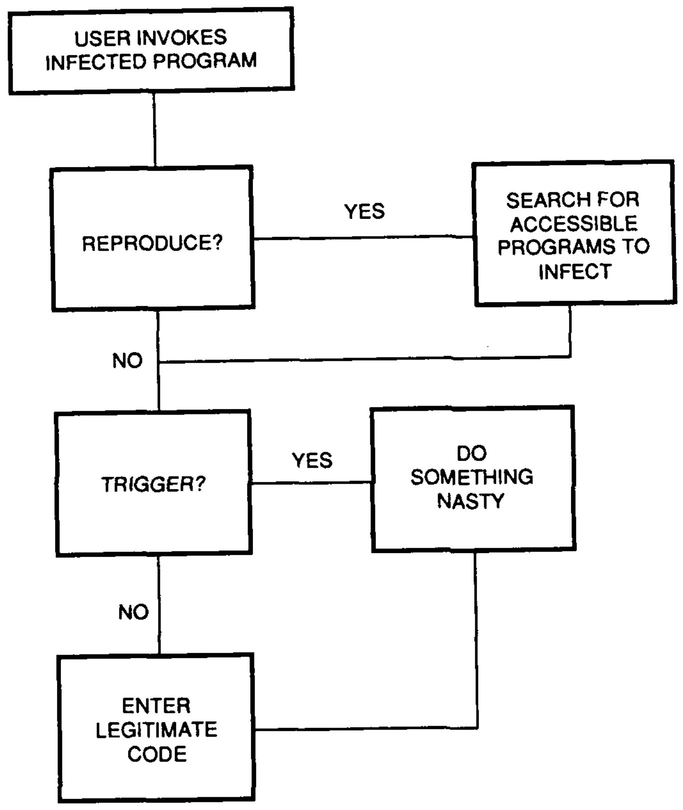
USER INVOKES INFECTED PROGRAM
REPRODUCE?
YES
SEARCH FOR ACCESSIBLE PROGRAMS TO INFECT
NO
TRIGGER?
YES
DO SOMETHING NASTY
NO
ENTER LEGITIMATE CODE

Figure 1. Flow control of a virus

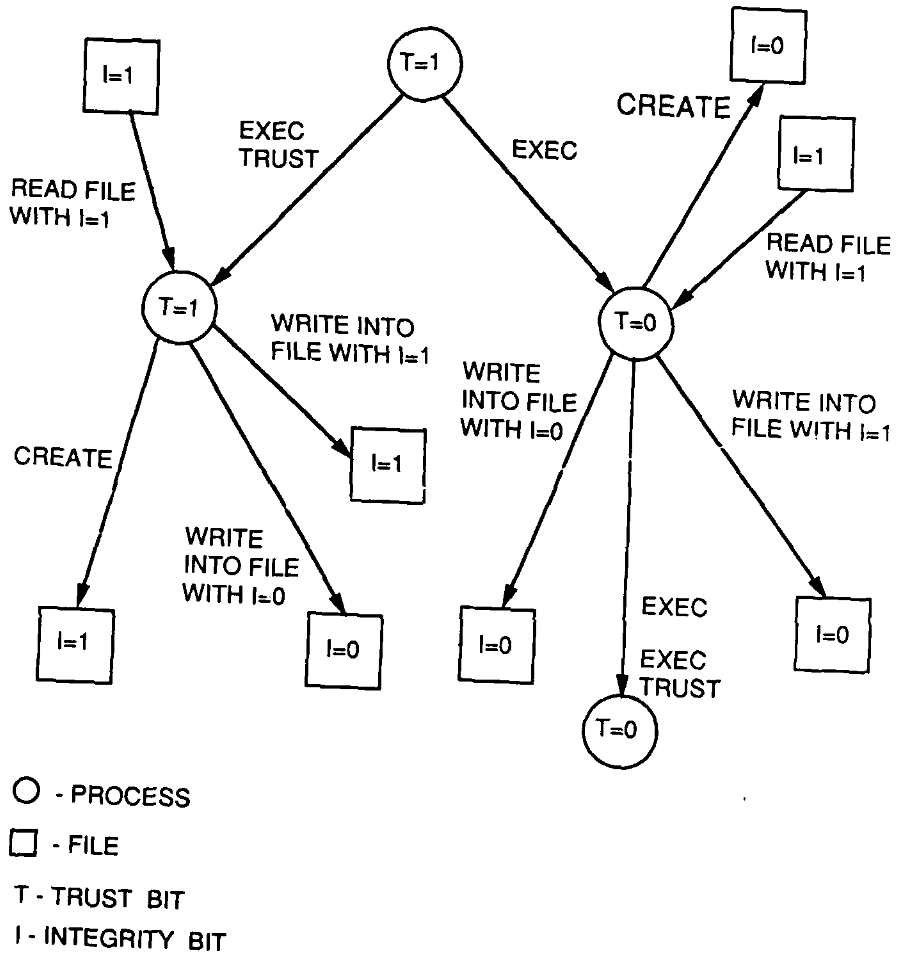

Figure 2. Limited integrity

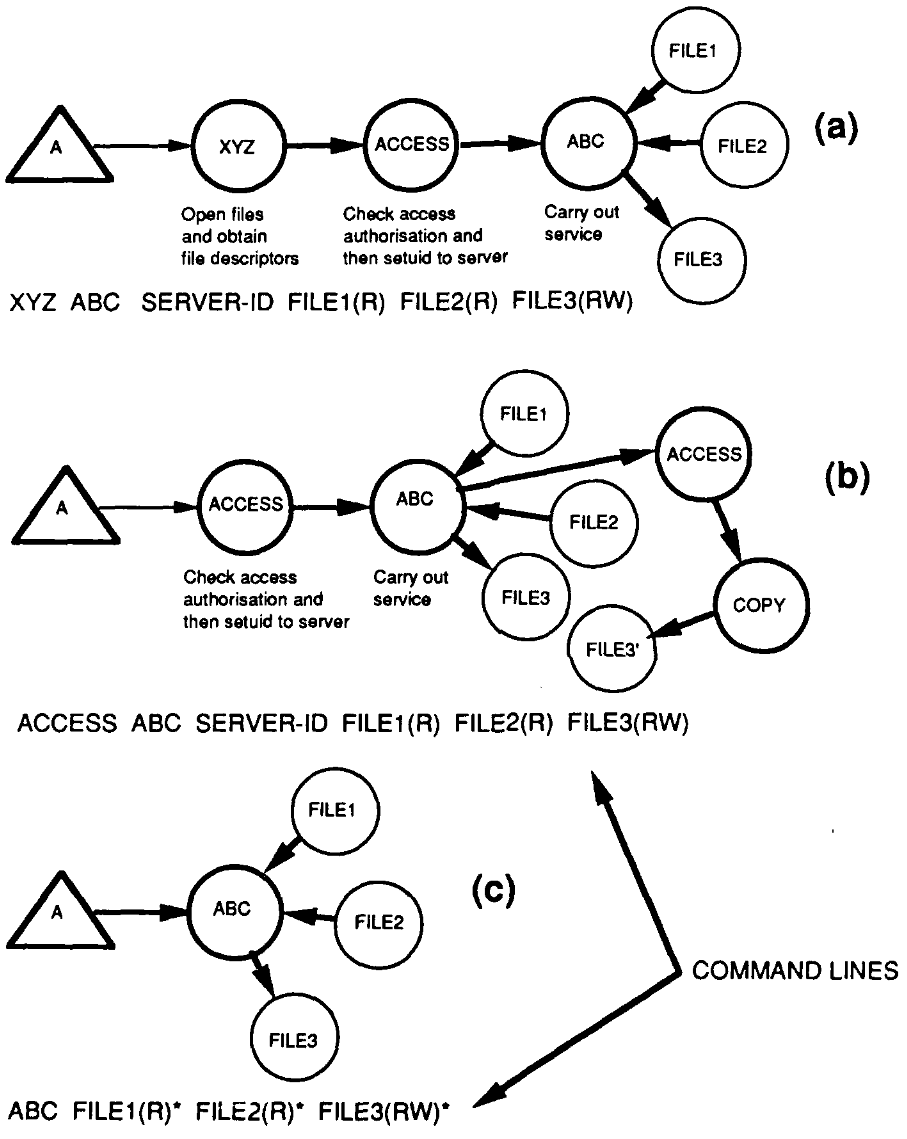
A
XYZ
ACCESS
ABC
FILE1
FILE2
FILE3
(a)
Open files and obtain file descriptors
Check access authorisation and then setuid to server
Carry out service
XYZ ABC SERVER-ID FILE1(R) FILE2(R) FILE3(RW)
A
ACCESS
ABC
FILE1
ACCESS
FILE2
FILE3
COPY
FILE3'
(b)
Check access authorisation and then setuid to server
Carry out service
ACCESS ABC SERVER-ID FILE1(R) FILE2(R) FILE3(RW)
A
ABC
FILE1
FILE2
FILE3
(c)
COMMAND LINES
ABC FILE1(R)* FILE2(R)* FILE3(RW)*

Figure 3. Invocation methods

DISTRIBUTION

	Number of copies
DEPARTMENT OF DEFENCE	
Defence Science and Technology Organisation	
Chief Defence Scientist	1
First Assistant Secretary Science (Policy)	
First Assistant Secretary Science Corporate Management	
Director General Science Technology Programs	
Director Joint Communications	1
Defence Signals Directorate Assistant Director, CS	1
Electronics Research Laboratory	
Director, Electronics Research Laboratory	1
Chief, Information Technology Division	1
Chief, Communications Division	1
Head, Software Engineering Group	1
Head, Trusted Computer Systems Group	1
Head, Information Systems Research Group	1
Head, Command Support Systems Group	1
PRS, Computer Architectures Research Team	1
PEVLSI, Research Team	1
Mr John Christie, Trusted Computer Systems	1

Dr M. Anderson, Trusted Computer Systems	1
N8014/2/1	1
Navy Office	
Director Management Informations Systems - Navy	1
Director Surface Warfare and Command Control - Navy	1
Navy Scientific Advisor	Cnt Sht Only
Army Office	
Director Joint Command and Control (HQADF)	1
Director of Command and Control Systems - Army	1
HQADF Scientific Advisor	1
Scientific Advisor - Army	1
Air Office	
Director Air Force Intelligence and Security	1
Air Force Scientific Adviser	1
Joint Intelligence Organisation (DSTI)	1
Libraries and Information Services	
Librarian, Technical Reports Centre, Defence Central Library, Campbell Park	1
Document Exchange Centre Defence Information Services Branch for:	
Microfiche copying	1
United Kingdom, Defence Research Information Centre	1
United States, Defense Technical Information Center	12
Canada, Director, Scientific Information Services	1

New Zealand, Ministry of Defence	1
National Library	1
Main Library, Defence Science and Technology Organisation Salisbury	2
Library, Aeronautical Research Laboratories	1
Library, Materials Research Laboratories	1
Librarian, DSD, Melbourne	1
Spares	2
Total number of copies	49

DOCUMENT CONTROL DATA SHEET

Security classification of this page : UNCLASSIFIED

1 DOCUMENT NUMBERS	**2 SECURITY CLASSIFICATION**
AR Number : AR-005-948	a. Complete Document : Unclassified
Series Number : ERL-0490-TR	b. Title in Isolation : Unclassified
Other Numbers :	c. Summary in Isolation : Unclassified
	3 DOWNGRADING / DELIMITING INSTRUCTIONS
	Limitation to be reviewed in September 1992

4 TITLE

SOME COMMENTS ON TECHNIQUES FOR RESISTING COMPUTER VIRUSES

5 PERSONAL AUTHOR (S)	**6 DOCUMENT DATE**
M. Anderson	September 1989
	7 7.1 TOTAL NUMBER OF PAGES 18
	7.2 NUMBER OF REFERENCES 12
8 8.1 CORPORATE AUTHOR (S)	**9 REFERENCE NUMBERS**
Electronics Research Laboratory	a. Task : DEF 89/183
	b. Sponsoring Agency :
8.2 DOCUMENT SERIES and NUMBER	**10 COST CODE**
Technical Report 0490	255
11 IMPRINT (Publishing organisation)	**12 COMPUTER PROGRAM (S)** (Title (s) and language (s))
Defence Science and Technology Organisation Salisbury	

13 RELEASE LIMITATIONS (of the document)

Approved for Public Release.

Security classification of this page : UNCLASSIFIED

Security classification of this page : UNCLASSIFIED

14 ANNOUNCEMENT LIMITATIONS (of the information on these pages)

No limitation

15 DESCRIPTORS

a. EJC Thesaurus Terms	Computer security Computer program integrity Computer programs	
b. Non - Thesaurus Terms	Computer viruses Computer worms	

16 COSATI CODES

0062B

17 SUMMARY OR ABSTRACT
(if this is security classified, the announcement of this report will be similarly classified)

Computer viruses have caused much concern. A basic flaw in the protection mechanism of most computer systems makes them very susceptible to viral attack. Various techniques for resisting viruses are discussed and their comparative advantages and disadvantages examined.

Security classification of this page : UNCLASSIFIED